Introduction to Generics

• Beginning with version 5.0, Java allows class and method definitions that include parameters for types
• Such definitions are called generics
  — Generic programming with a type parameter enables code to be written that applies to any class
• Already have been using generics
  — E.g. `ArrayList<BaseType> aList = new ArrayList<BaseType>();`

Defining a Class with a Type Parameter

```java
Display 14.4 A Class Definition with a Type Parameter
1  public class Sample<T> {
2    private T data;
3    
4    public void setData(T newData) {
5        data = newData;  // T is a parameter for a type.
6    }
7    
8    public T getData() {
9        return data;
10    }
11 }
12 }
```
Class Definition with a Type Parameter

• A class that is defined with a parameter for a type is called a generic class or a parameterized class
  – The type parameter is included in angular brackets after the class name in the class definition heading
  – Any non-keyword identifier can be used for the type parameter, but by convention, the parameter starts with an uppercase letter
  – The type parameter can be used like other types used in the definition of a class

Tip: Compile with the \(-\text{Xlint}\) Option

• There are many pitfalls that can be encountered when using type parameters
• Compiling with the \(-\text{Xlint}\) option will provide more informative diagnostics of any problems or potential problems in the code
  \texttt{javac -Xlint Sample.java}
A Generic Ordered Pair Class
(Part 1 of 4)

```java
public class Pair<T> {
    private T first;
    private T second;

    public Pair() {
        first = null;
        second = null;
    }

    public Pair(T firstItem, T secondItem) {
        first = firstItem;
        second = secondItem;
    }

    // (continued)
```

A Generic Ordered Pair Class
(Part 2 of 4)

```java
public void setFirst(T newFirst) {
    first = newFirst;
}

public void setSecond(T newSecond) {
    second = newSecond;
}

public T getFirst() {
    return first;
}

// (continued)
```
A Generic Ordered Pair Class
(Part 3 of 4)

Display 14.5  A Generic Ordered Pair Class

```java
public T getSecond()
{
    return second;
}

public String toString()
{
    return ("first: " + first.toString() + "\n" + "second: " + second.toString());
}
```

(continued)

A Generic Ordered Pair Class
(Part 4 of 4)

```
public boolean equals(Object otherObject)
{
    if (otherObject == null)
        return false;
    else if (getClass() != otherObject.getClass())
        return false;
    else
    {
        Pair<T> otherPair = (Pair<T>)otherObject;
        return (first.equals(otherPair.first) && second.equals(otherPair.second));
    }
}
```

14-7
Using Our Ordered Pair Class
(Part 1 of 3)

1 import java.util.Scanner;

2 public class GenericPairDemo
3 {
4     public static void main(String[] args)
5     {
6         Pair<String> secretPair =
7             new Pair<String>("Happy", "Day");
8     
9         Scanner keyboard = new Scanner(System.in);
10        String word1 = keyboard.next();
11        String word2 = keyboard.next();
12        Pair<String> inputPair =
13            new Pair<String>(word1, word2);
14
15        if (inputPair.equals(secretPair))
16        {
17            System.out.println("You guessed the secret words");
18            System.out.println("In the correct order!");
19        }
20        else
21        {
22            System.out.println("You guessed incorrectly.");
23            System.out.println("You guessed ");
24            System.out.println(inputPair);
25            System.out.println("The secret words are");
26            System.out.println(secretPair);
27        }
28    }
29}

(continued)
Using Our Ordered Pair Class (Part 3 of 3)

**Pitfall: A Generic Constructor Name Has No Type Parameter**

- Although the class name in a parameterized class definition has a type parameter attached, the type parameter is not used in the heading of the constructor definition
  
  ```java
  public Pair<T>()
  ```

- A constructor can use the type parameter as the type for a parameter of the constructor, but in this case, the angular brackets are not used
  
  ```java
  public Pair(T first, T second)
  ```

- However, when a generic class is instantiated, the angular brackets are used
  
  ```java
  Pair<String> pair =
  new Pair<String>("Happy", "Day");
  ```
Pitfall: A Primitive Type Cannot be Plugged in for a Type Parameter

• The type plugged in for a type parameter must always be a reference type
  – It cannot be a primitive type such as `int`, `double`, or `char`
  – However, now that Java has automatic boxing, this is not a big restriction
    • Use `Integer`, `Double`, `Char`
  – Note: reference types can include arrays

Pitfall: A Type Parameter Cannot Be Used Everywhere a Type Name Can Be Used

• Within the definition of a parameterized class definition, there are places where an ordinary class name would be allowed, but a type parameter is not allowed
• In particular, the type parameter cannot be used in simple expressions using `new` to create a new object
  – For instance, the type parameter cannot be used as a constructor name or like a constructor:
    ```java
    T object = new T();
    T[] a = new T[10];
    ```
Pitfall: An Instantiation of a Generic Class
Cannot be an Array Base Type

• Arrays such as the following are illegal:

\[
\text{Pair<String>[]} a = \\
\text{new Pair<String>}[10];
\]

— Although this is a reasonable thing to want to do, it is not allowed given the way that Java implements generic classes

Using Our Ordered Pair Class and Automatic Boxing (Part 1 of 3)

```
import java.util.Scanner;

public class GenericPairDemo
{
    public static void main(String[] args)
    {
        Pair<Integer> secretPair = 
            new Pair<Integer>(42, 24);

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter two numbers:");
        int n1 = keyboard.nextInt();
        int n2 = keyboard.nextInt();
        Pair<Integer> inputPair = 
            new Pair<Integer>(n1, n2);
    }
}
```
Using Our Ordered Pair Class and Automatic Boxing (Part 2 of 3)

Display 14.7 Using Our Ordered Pair Class and Automatic Boxing

```java
if (inputPair.equals(secretPair))
{
    System.out.println("You guessed the secret numbers");
    System.out.println("in the correct order!");
}
else
{
    System.out.println("You guessed incorrectly.");
    System.out.println("You guessed");
    System.out.println(inputPair);
    System.out.println("The secret numbers are");
    System.out.println(secretPair);
}
``` (continued)

Using Our Ordered Pair Class and Automatic Boxing (Part 3 of 3)

Display 14.7 Using Our Ordered Pair Class and Automatic Boxing

SAMPLE DIALOGUE

Enter two numbers:
42 24
You guessed the secret numbers
in the correct order!
Pitfall: A Class Definition Can Have More Than One Type Parameter

- A generic class definition can have any number of type parameters
  - Multiple type parameters are listed in angular brackets just as in the single type parameter case, but are separated by commas

```java
public class TwoTypePair<T1, T2> {
    private T1 first;
    private T2 second;

    public TwoTypePair() {
        first = null;
        second = null;
    }

    public TwoTypePair(T1 firstItem, T2 secondItem) {
        first = firstItem;
        second = secondItem;
    }
}
```
Multiple Type Parameters (Part 2 of 4)

```java
public void setFirst(T1 newFirst)
{
    first = newFirst;
}

public void setSecond(T2 newSecond)
{
    second = newSecond;
}

public T1 getFirst()
{
    return first;
}

(continued)
```

Multiple Type Parameters (Part 3 of 4)

```java
public T2 getSecond()
{
    return second;
}

public String toString()
{
    return ("first: " + first.toString() + "\n" + "second: " + second.toString());
}

(continued)
```
Multiple Type Parameters (Part 4 of 4)

Display 14.8  Multiple Type Parameters

37    public boolean equals(Object otherObject)  
38    {  
39      if (otherObject == null)  
40        return false;  
41      else if (getClass() != otherObject.getClass())  
42        return false;  
43      else  
44        {  
45          TwoTypePair<T1, T2> otherPair = (TwoTypePair<T1, T2>)otherObject;  
46          return (first.equals(otherPair.first)  
47                     & second.equals(otherPair.second));  
48        }  
49    }  
50
The first equals in the equals of the type T1. The second equals in the equals of the type T2.

Using a Generic Class with Two Type Parameters (Part 1 of 2)

Display 14.9  Using a Generic Class with Two Type Parameters

1    import java.util.Scanner;
2    
3    public class TwoTypePairDemo  
4    {  
5      public static void main(String[] args)  
6      {  
7          TwoTypePair<String, Integer> rating = new TwoTypePair<String, Integer>("The Cor Guys", 8);
8          Scanner keyboard = new Scanner(System.in);
9          System.out.println("Our current rating for " + rating.getFirst());  
10         System.out.println(" is ", + rating.getSecond());  
11         System.out.println("How would you rate them?");  
12         int score = keyboard.nextInt();  
13         rating.setSecond(score);
14       (continued)
Pitfall: A Generic Class Cannot Be an Exception Class

- It is not permitted to create a generic class with Exception, Error, Throwable, or any descendent class of Throwable
  - A generic class cannot be created whose objects are throwable
    public class GEx<T> extends Exception
  - The above example will generate a compiler error message
Bounds for Type Parameters

• Sometimes it makes sense to restrict the possible types that can be plugged in for a type parameter $T$
  – For instance, to ensure that only classes that implement the `Comparable` interface are plugged in for $T$, define a class as follows:
    ```java
    public class RClass<T extends Comparable>
    ```
  – "extends Comparable" serves as a bound on the type parameter $T$
  – Any attempt to plug in a type for $T$ which does not implement the `Comparable` interface will result in a compiler error message

• A bound on a type may be a class name (rather than an interface name)
  – Then only descendent classes of the bounding class may be plugged in for the type parameters
    ```java
    public class ExClass<T extends Class1>
    ```

• A bounds expression may contain multiple interfaces and up to one class

• If there is more than one type parameter, the syntax is as follows:
    ```java
    public class Two<T1 extends Class1, T2 extends Class2 & Comparable>
    ```
A Bounded Type Parameter

Tip: Generic Interfaces

- An interface can have one or more type parameters
- The details and notation are the same as they are for classes with type parameters
Generics and Methods

• Consider writing a method that takes an array of objects and a collection and puts all objects in the array into the collection. Here's a first attempt (that doesn't work):

```java
static void fromArrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o); // Compile time error
    }
}
```

Can't put objects of an unknown type into the Collection

Solution - Generic Methods

• When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class
• In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class
  – A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter
  – The type parameter of a generic method is local to that method, not to the class
Generic Methods

• The type parameter must be placed (in angular brackets) after all the modifiers, and before the returned type
  
  ```java
  public static <T> T genMethod(T[] a)
  ```

• When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angular brackets (if left off, Java will infer the type)
  
  ```java
  String s = NonG.<String>genMethod(c);
  ```

Generic Collection Example

```java
static <T> void fromArrayToCollection(T[] a, Collection<T> c)
{
    for (T o : a)
    {
        c.add(o); // Correct
    }
}

fromArrayToCollection(strArray, strCollection)
```
Inheritance with Generic Classes

• A generic class can be defined as a derived class of an ordinary class or of another generic class
  – As in ordinary classes, an object of the subclass type would also be of the superclass type
• Given two classes: A and B, and given G: a generic class, there is no relationship between G<A> and G<B>
  – This is true regardless of the relationship between class A and B, e.g., if class B is a subclass of class A

A Derived Generic Class (Part 1 of 2)

Display 14.11 A Derived Generic Class

```java
public class UnorderedPair<T> extends Pair<T>
{
    public UnorderedPair()
    {
        setFirst(null);
        setSecond(null);
    }

    public UnorderedPair(T firstItem, T secondItem)
    {
        setFirst(firstItem);
        setSecond(secondItem);
    }
}
```

(continued)
A Derived Generic Class (Part 2 of 2)

Display 14.11  A Derived Generic Class

```java
13  public boolean equals(Object otherObject) {
14      if (otherObject == null) {
15          return false;
16      } else if (getClass() != otherObject.getClass()) {
17          return false;
18      } else {
19          UnorderedPair<T> otherPair = (UnorderedPair<T>) otherObject;
20          return (getFirst().equals(otherPair.getFirst())) &&
21                 (getSecond().equals(otherPair.getSecond()));
22      }
23  }
24  }
```

Using UnorderedPair (Part 1 of 2)

Display 14.12  Using UnorderedPair

```java
1  public class UnorderedPairDemo {
2  {
3      public static void main(String[] args) {
4          UnorderedPair<String> p1 =
5              new UnorderedPair<String>("peanuts", "beer");
6          UnorderedPair<String> p2 =
7              new UnorderedPair<String>("beer", "peanuts");
8          (continued)
```
Using UnorderedPair (Part 2 of 2)

```java
if (p1.equals(p2)) {
    System.out.println(p1.getFirst() + " and " + p1.getSecond() + " is the same as " + p2.getSecond());
}
```

**Sample Dialogue**

- Peanuts and beer is the same as beer and peanuts